Pre-02-Verilog

1.Verilog建模概述

1.1.结构级建模

- 简介：将模块用导线连接起来形成的一个功能元件
- 实例化：模块名 实例名(端口映射)
  - 端口映射
    1. `(信号1, 信号2, ……, 信号n)` 变量i对应第i个端口
    2. `(.端口名1(信号1), .端口名2(信号2), ..., .端口名n(信号n))`
  - 注意：实例化元件时，wire型可以连接到任何端口，而reg型只能连接到输入端口；声明元件时，reg型只能连接到输出端口

1.2.行为级描述

- 简介：侧重输入和输出之间的关系，描述行为操作
- 连续赋值语句assign
  - 左边必须是wire型，assign将右边的值时时刻刻赋到左边
  - 注意：assign不可在always块中使用
- 过程控制语句
  - 场景: 只能出现在initial块和always块
  - 两个常见语句块
    - `initial`块
      - 初始化，从仿真0时刻开始，只执行1次
    - `always`块
      - 边沿敏感（时序电路）（阻塞赋值）
      - 电平敏感（组合电路）（非阻塞赋值）
  - 常见过程控制语句
    - `if`

```
if ()
    begin
        /* code */
    end
else if()
    begin
        /* code */
    end
else
    begin
        /* code*/
    end
```

  - `while`

```
while ()
    begin
        /* code */
    end
```

- `for`

```
for(expression1; expression2; expression3)
    begin
        /* code */
    end
```

2.语法

2.1.模块

模块的定义有两种方式：

```
//the first way
module AndGate{
    input inA,
    input inB,
    output ouC
};
    assign ouC = inA & inB;
endmodule
//the second way
module AndGate{inA, inB, ouC};
    input inA;
    input inB;
    output ouC;
    assign ouC = inA & inB;
endmodule
```

被测文件中，输入必须是wire型；测试文件（tb）中，输出必须是wire型。

2.2.常见数据类型

2.2.1.wire

隶属net型。没有输入就没有输出，必须依靠assign语句驱动。参与组合逻辑。

wire有两种类型：标量（1位）、矢量（多位）。声明多位wire型时，应使用范围声明器，如`wire[31:0] a`。访问时可以使用范围声明器访问变量的局部位，如`a[7:4]`取出a的第7到4位数据。使用wire前必须声明。如果声明变量时并没有标明类型，默认为1位wire型。

> **对矢量的说明**
>
> 位宽和方向。例如定义为[4:7]的b信号，四个管脚4、5、6、7，在使用中只能正向接，不能反向接。所以o[3:0] = b[4:7]是合法的，而o[3:0] = b[7:4]不合法。

2.2.2.reg

具有存储功能。只在initial块和always块中使用。reg类型并不一定综合为寄存器。

范围声明部分同wire型。

> **利用reg类型建模存储器**
>
> 可以利用reg型产生数组。声明方法：`reg[31:0] mem[0:1023]`前中括号代表位宽，后中括号代表存储器数量。
>
> 可以通过引用操作访问存储器型元素，类似于位访问。例如`mem[2]`，即访问mem数组中的第三个元素。

2.2.3.数字字面量

- 声明方式：`[data bits]'[radix][data]`
- 注意事项
  - 表示负数的时候，应将符号写在data bits前
  - 部分位可以用x或z表示。x：不定值，当某位的值不能确定时，用x表示。z：高阻态，代表没有连接到有效输入。
  - 进制缺省时默认十进制
  - data部分可以用下划线分开提高可读性
  - data部分不可超过位宽限制
- 示例：

```
2'b11    //二进制 3
3'5      //十进制 5
-4'o11   //十进制 -11
8'h101z  //十六进制
```

2.2.4.integer

默认32位，有符号数。**实验中主要用于for循环。**

2.2.5.parameter

`parameter 标识符 = 表达式;`用于编译时确认值的常量。

2.3.常用运算符

- 逻辑右移运算符`>>`算数右移运算符`>>>`
- 相等比较运算符`==`、`!=` `===`、`!===`

  前者可能由于x和z的出现，出现结果为x；而后者将x和z也列入比较，结果一定是0/1
- 位拼接运算符`{}`
  - 可以将几个信号的某些位拼接起来，例如`{a[4:1], b, 3'b101}`
  - 可以简化重复的表达式，例如`{4{w}}`等价于`{w,w,w,w}`，`{b, 3{a,b}}`等价于`{b, {a,b,a,b,a,b}}`，等价于`{b,a,b,a,b,a,b}`

- 缩减运算符

  `&`，`|`，`^`等作为单目运算符，表示对操作数的每一位汇总运算

- 阻塞赋值与非阻塞赋值

  为reg型变量赋值

2.4.组合逻辑建模方式

- assign连续赋值语句：assign只驱动wire
- 电平敏感的always块
  - 被赋值的变量一定要定义为reg型
  - 敏感条件必须包含块内所有输入信号，也就是`@*`或`@(*)`，否则会产生意想不到的latch
  - 块内else(if-else)和default(case)必须补充完整，否则也会产生latch
  - 不要出现任何有存储意义的语句

    此点仅限于组合逻辑。以下举例说明：

    ```
    //时序电路
    always @(posedge clk)begin
        if(enable)
            q <= data;
    end
    //组合电路
    always @(*)begin
        if(enable)
            q = data; //此处用阻塞赋值/非阻塞赋值，从结果上看没什么差别
    end
    ```

    时序电路生成触发器。触发器自带使能端，当使能端处于低电平时，触发器可以保存数据，无需latch。

    组合电路生成锁存器。当使能信号处于低电平时，输出要保持不变，而组合逻辑并没有存储元件，只能生成latch来保存数据。

    针对这个组合电路生成latch的例子，解决方法：1.改成时序电路；2.补充完整else；3.输出初始化，如下代码：

    ```
    always @* begin
        q = 0;
        if(enable)
            q = data;
    end
    ```

2.4.1.latch

2.4.1.1.特点

- 对毛刺敏感

  使能信号处于有效电平时，输出状态可能随输入多次变化，产生空翻，不能**异步复位**，因此**在上电后处于不确定的状态**

- 使静态时序分析变得复杂，不具备可重用性
  - latch没有时钟参与信号传递，无法做STA

    - 综合工具会将latch优化，造成前后仿真效果不一致
- 生成锁存器需要更多的资源

  如果组合逻辑的语句完全不使用always语句块，就可以保证综合器不会综合出锁存器

2.4.1.2.出现

- 在赋值表达式右边参与赋值的信号未在always@(敏感电平列表)中列出完整，生成透明锁存器
- else和default不完整
- 输出变量赋值给自己且无初始化

2.4.1.3.对应的解决方法

- 检查补齐敏感列表，最好使用@*
- 补全if-else和case
- **在不影响设计需求的情况下**，给输出变量赋初始值

2.5.时序逻辑建模方式

- 边沿敏感的always块
    - 同步电路`always @(posedge clk)`
    - 异步电路`always @(posedge clk or negedge rst)`：块内第一行必须是异步输入
- 注意：时序电路生成触发器，有保存数据的功能。因此和锁存器无缘。

2.6.语句块

2.6.1.顺序快

以`begin-end`为开头结尾的代码块，也就是说从细节上理解，其中代码是按顺序进行的

2.6.2.并行块

以`fork-join`为开头结尾的代码块

2.7.过程块

2.7.1.always块

- 组合逻辑-电平敏感

  `always @(敏感条件列表)`(注意补齐列表) 或 `always@*` 或 `always@(*)`
- 时序逻辑-边沿敏感
    - 同步电路`always @(posedge clk)`
    - 异步电路`always @(posedge clk or negedge rst)`：块内第一行必须是异步输入
- 时间控制
    - ```
      always #5 b = a;
      ```
      ```

2.7.2.initial块

仅在仿真中使用，**不可综合**

2.8.过程赋值语句

2.8.1.阻塞赋值(=)

- 阻塞赋值语句**立即**执行，执行**完毕**后才执行下一个语句。
- 左侧值在赋值语句执行后**立即**改变

2.8.2.非阻塞赋值(<=)

- 语句执行到此，先计算右侧值，但**不立即赋值**给左侧
- always**块结束后**才完成此次赋值操作
- Use $strobe to display values that have been assigned using nonblocking assignments.

2.8.3.选用赋值方式

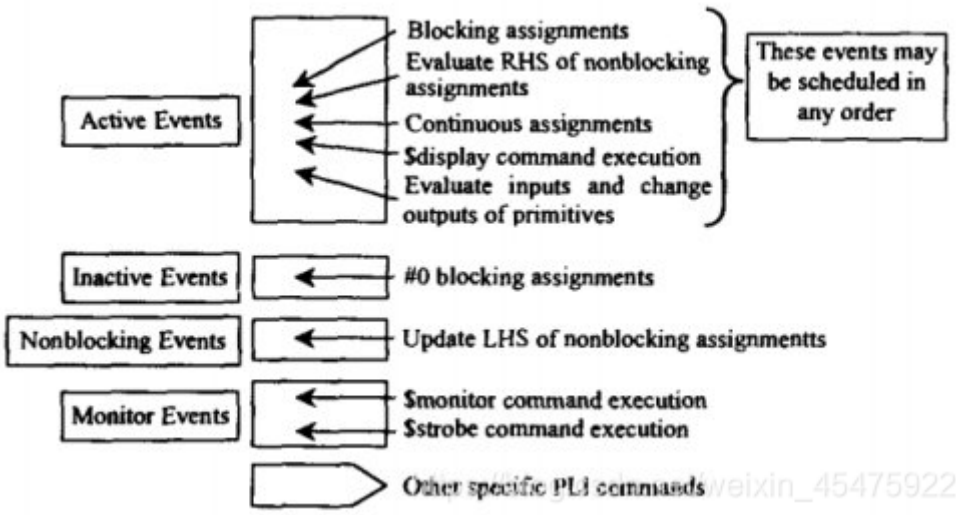
- always块纯组合逻辑模型：阻塞赋值
- 时序电路或锁存器建模：非阻塞赋值
- 在同一个always块中建立时序和组合逻辑电路：非阻塞赋值
- 在同一个always块中不能两种方式混用

2.8.4.层次化队列

verilog将所有事件分为4个优先级，优先级相同的事件为1个队列，优先级高的队列的事件全部执行完毕后才会执行优先级低队列的事件。优先级顺序如下（由高到低）：

1. 动态事件队列
    - 解释：动态事件队列在队列内部的执行**顺序并没有硬性规定。**但是，在**同一个**`begin-end`语句块中的语句应当严格按照源代码中的**顺序执行**；且多个非阻塞赋值应当按照语句执行顺序进行。
    - 下列事件的顺序可以随意调换：
        - 阻塞赋值
        - 计算非阻塞赋值语句右边的表达式（RHS）
        - 连续赋值（如`assign`）
        - 执行`$display`命令
        - 执行原语的输入和输出的变化
2. 停止运行的时间队列（`#0`）（不推荐使用）
3. 非阻塞事件队列：更新非阻塞赋值语句LHS的值

4. 监控事件队列（执行`$monitor`, `$strobe`命令）



- 探究

  第9行和第10行谁先执行?

```
1  module event_horizen(
2      input a,
3       input clk,
4       output reg b,
5      output reg[2:0] c
6      );
7
8      always @(posedge clk) begin
9          b <= a;
10         if (b == 1) begin
11             c <= 3'b010;
12         end else begin
13             c <= 3'b011;
14         end
15     end
16
17 endmodule
```

  testbench代码:

```
1  module event_horizen_tb;
2
3      // Outputs
4      wire[2:0] c;
5      wire b;
6      reg a;
7      reg clk;
8
9      // Instantiate the Unit Under Test (UUT)
10     event_horizen uut (
11        .clk(clk),
12         .c(c),
13         .b(b),
14         .a(a)
15     );
16
```

```
17      initial begin
18          // Initialize Inputs
19          clk = 0;
20          a = 1;
21          // Wait 100 ns for global reset to finish
22          #50;
23      end
24
25      always #5 clk = ~clk;
26  endmodule
```

如图，第9行非阻塞赋值的RHS执行 -> if块执行 -> always块结束后LFS赋值执行。

```
20  //////////////////////////////////////////////////////////////////////////////////
21  module event_horizen(
22      input a,
23      input clk,
24      output reg b,
25      output reg[2:0] c
26      );
27
28      always @(posedge clk) begin
29          b <= a;
30          if (b == 1) begin
31              c <= 3'b010;
32          end else begin
33              c <= 3'b011;
34          end
35      end
```

2.9.过程控制语句

- if、case

  只用于顺序块

  - case、casex和casez（据说，少用casex）

case	0	1	x	z	casez	0	1	x	z	casex	0	1	x	z
0	1	0	0	0	0	1	0	0	1	0	1	0	1	1
1	0	1	0	0	1	0	1	0	1	1	0	1	1	1
x	0	0	1	0	x	0	0	1	1	x	1	1	1	1
z	0	0	0	1	z	1	1	1	1	z	1	1	1	1

- for
  - 循环变量
    - **integer**

        - reg: 特别注意其**位宽**，以防进入死循环状态
    - 循环结束条件
        - 循环上限使用**常量**，例如`i < const`。若使用变量容易造成死循环
        - 即使循环上限必须有变量的参与，也要加入常量的限制。例如，inputA限制最大值为31，建议写`i < inputA && i < 32`
    - for语句对应实际线路：建议想好实际线路之后再写代码（感觉有点抽象……）

2.A.时间控制语句

- 类型
    - 外部延时

      当RHS变量有变化时，等待#inter_delay时间后评估RHS的值，并赋给LHS变量

      ```
      #inter_delay;
      b = a;

      #inter_delay b = a;
      assign #inter_delay b = a;
      ```

    - 内部延时

      当RHS变量有变化时，立刻评估RHS的结果，延迟#intra_delay时间后，将评估的结果赋给LFS

      注：只能用在**顺序**语句中

      ```
      b = #intra_delay a;
      ```

- 惯性延迟

  当输入较小宽度的脉冲将会被滤除，即**不允许所有宽度小于指定延迟的脉冲通过电路单元。**故为了使器件对输入信号的变化产生变化，信号变化后要维持足够的时间。

  因此仿真时，时延一定要**合理设置**，防止某些信号不能进行有效的延迟。

2.B.有符号数的处理方法

2.B.1.数据类型的符号

- 无符号数：wire，reg。使用`$signed()`进行向符号数的转换
- 有符号数：integer

2.B.2.原理

- verilog处理有符号数的流程
    1. 确定最外层表达式的符号
        - 自决定表达式
        - 关系表达式与等式表达式

          结果是自决定的，但**子表达式要互相影响。**总体而言，介于自决定和上下文之间。比如，`a > $signed(b)`仍然进行向内传播，也就是当作`a > b`进行运算。

        - 上下文决定表达式

其符号取决于子表达式。因此应将表达式写成树的格式。从树的叶节点（原子表达式）往上推。

简单概括：若表达式中有任何一项是无符号的，那么表达式就是无符号的。

- 特例：
  - 移位运算符的右侧总是被视为无符号数，而且对运算结果的符号性没有影响
  - 三目运算符：其?前的布尔表达式是自决定的表达式，不会对最外层表达式的符号造成影响。

2. 向内传播

将最外层表达式的位宽和符号由外向内地传递给上下文决定的子表达式（自决定的子表达式不受影响），直到遇到原子表达式，此时就要**将原子表达式进行强制类型转换。**

2.8.3.推荐使用方案总结

- 表达式中的**运算数的符号类型最好一致**

  使用`$signed()`和`$unsigned()`函数统一符号类型。不要在表达式中混用符号数。

- 复杂的表达式中，可以把`$signed()`单独抽离出来作为变量
- 尽量避开`$signed()`

  例如，可以使用位拼接运算符。

3.高级特性

3.1.编译预处理

1. `` `define `` 宏定义

   声明和C类似，注意**在引用宏名的时候也要加上`**

   勿滥用。若宏内容较长且在代码中使用次数较多，在线评测可能会发生**编译超时。**

2. `` `inculde `` 文件包含
   - 含义：一个源文件可以通过此命令将另一个源文件的全部内容包含进来。其一般形式为：

     ```
     `include "filename"
     ```

   - 编译过程，将File2.v的全部内容复制插入到该命令出现的地方。
   - 使用方法
     - 绝对路径：万无一失但相对麻烦
     - 相对路径
     - 文件名：需要将文件放在同一个目录下
   - 说明
     - 一个`` `include ``命令只能包含一个文件。若想包含多个文件，必须使用多次命令
     - 该命令可以出现在Verilog HDL源程序的任何地方。被包含文件名可以是相对路径名，也可以是绝对路径名
     - 优先级问题：书写顺序在后的文件默认可以使用顺序在前的文件内容
     - 不要inculde包含模块定义的文件，而是将定义和使用模块的.v文件全部加入工程中，一起编译。否则在线评测时可能造成“模块重复定义”错误
3. `` `timescale `` 时间尺度

```
`timescale [时间单位]/[时间精度]
//如下例，此命令后，模块中所有的时间值都表示为1ns的整数倍
`timescale 1ns/1ps;
```

4. 条件编译语句

```
`ifdef 宏名a
    程序段1
`elsif 宏名b
    程序段2
`else
    程序段3
`endif
```

当宏名a被定义时，执行程序段1；当宏名a不被定义且宏名b被定义时执行程序段2；否则执行程序段3

与ifdef相对的是`ifndef`

3.2.系统任务

3.2.1.输出信息

```
$display(p1, p2, ..., pn);
```

- 输出格式

输出格式	说明
%h or %H	十六进制
%d or %D	十进制
%b or %B	二进制
%c or %C	ASCII码字符
%s or %S	字符串

- 示例

```
$display("hello world");
$display("hello", "world");
$display("hello, my id is %h", student_id);
$display("hello, the time is %t", time); //时间的控制符不同
```

3.2.2.监控变量

- `$monitor(p1, p2, ..., pn);`
  - 功能：监控和输出参数列表中的表达式或变量值
  - 参数列表输出控制格式字符串与输出列表的规则和`$display`一样
- `$monitor;`
- `$monitoron;` 打开监控标志，启动监控任务。$monitor往往在initial块中调用
- `$monitoroff;` 关闭监控标志，中断监控任务

3.2.3.读写内存 $readmemh()

应用情景：将16进制机器码导入ISE工程中。

参数：数据文件路径、目标寄存器、起始地址(optional)和终止地址(optional)

用法举例：

下面要将code.txt的数据加载进my_memory

```
$readmemh("code.txt", my_memory);
$readmemh("code.txt", my_memory, 128);
$readmemh("code.txt", my_memory, 128, 1);
```

3.3.缺省变量设置

不做任何设置时（不对某个变量显式声明类型或显式定义），缺省变量自定义为wire型。

通过下述语句可以设置代码文件所有变量的缺省类型：

```
`default_nettype 类型
```

如果将类型设置为none，则能够取消缺省类型，即若不显式声明即报错，如下：

```
`default_nettype none
```

3.4.函数

- 声明

  函数在模块中定义，位置任意，并在模块的任意地方被引用。用function声明，用endfunction结束，不允许输出端口声明（包括输出和双向端口），可以有多个输入端口，函数只返回1个值到函数被调用的位置，且**在函数中返回值与函数名同名。**

  定义如下：

  ```
  function [range] function_id;
      input_declaration
      other_declaration
      procedural_statement
  endfunction
  ```

- 注意
  - 函数定义只能在**模块**中完成。不能出现在过程块中；
  - 函数至少要有1个输入端口。不能有输出端口和双向端口；
  - 函数中**不能**使用时间控制语句（#，wait等），也不能使用disable中止语句
  - 函数定义结构体中**不能**出现过程块语句（always语句）
  - 函数内部可以调用函数
  - 函数内部**不能**调用任务
- 函数调用的注意事项
  - 函数调用可以在过程块和连续赋值语句中出现
  - 函数调用语句只能作为**赋值语句的RHS**，不可单独作为一条语句出现

4.ISE用法

4.1.查看中间变量

`Instances and Processes -> moduleName -> double click uut`：`Object` -> 选中需要的变量（甚至它的某位）->右键添加到波形窗口/`Ctrl + W`。

当有多个模块时，波形只显示顶层模块的IO端口，也采用如上的方式查看指定变量的波形。

4.2.仿真文件保存

为了方便再次仿真，可以保存Isim波形文件，命名并选好位置。

再次仿真时，Isim不会直接调用已经保存的仿真文件。需要手动打开。

4.3.断点调试

Instances and Processes Name ->double click moduleName（即打开对应文件） -> 在需要的地方加入断点 -> 点击reset（Ctrl + Shift + F5）-> run all（F5） （运行到断点处）-> step (F11)

4.4.Memory

- 若没有Memory，则View -> Panels -> Memory
- 可以修改Radix
- 若搜索某地址的值，则在地址栏中输入地址，回车

5.杂谈

5.1.FSM

5.1.1.善用“宏”

`` `define ``: 使用时需要在量名前加上`,较麻烦。但可以配合include语句达成多个文件的一致性。

`parameter`： 静态变量。但只能用于一个文件。

`wire`： 三目运算符式

```
1  `define limit0 "0"
2
3  module test(
4      input [7:0] char
5      );
6
7      parameter YES = 1'b1;
8      parameter NO = 1'b0;
9      wire digit = (char >= `limit0 && char <= "9") ? YES : NO;
10
11     /** statement **/
12
13 endmodule
```

5.1.2.状态转换的描述

一般用case或casex语句！**default不可或缺。**